

# The Impact of Software Evolution on Code Coverage Information

Sebastian Elbaum  
Dept. of Computer  
Science and Engineering  
University of Nebraska-Lincoln  
Lincoln, Nebraska  
elbaum@cse.unl.edu

David Gable  
Dept. of Computer  
Science and Engineering  
University of Nebraska-Lincoln  
Lincoln, Nebraska  
dgable@cse.unl.edu

Gregg Rothermel  
Computer Science Dept.  
Oregon State University  
Corvallis, OR  
grother@cs.orst.edu

## Abstract

*Many tools and techniques for addressing software maintenance problems rely on code coverage information. Often, this coverage information is gathered for a specific version of a software system, and then used to perform analyses on subsequent versions of that system without being recalculated. As a software system evolves, however, modifications to the software alter the software's behavior on particular inputs, and code coverage information gathered on earlier versions of a program may not accurately reflect the coverage that would be obtained on later versions. This discrepancy may affect the success of analyses dependent on code coverage information. Despite the importance of coverage information in various analyses, in our search of the literature we find no studies specifically examining the impact of software evolution on code coverage information. Therefore, we conducted empirical studies to examine this impact. The results of our studies suggest that even relatively small modifications can greatly affect code coverage information, and that the degree of impact of change on coverage may be difficult to predict.*

## 1 Introduction

Many software maintenance techniques and tools require knowledge about the dynamic behavior of software. Program profiling techniques [2, 15] provide such knowledge, collecting *code coverage information* about such things as the statements, branches, paths, or functions encountered or taken during a program's execution. Such code coverage information supports maintenance-related activities such as impact analysis [3, 8], dynamic slicing [1, 12, 14], assessments of test adequacy [16, 19], selective regression testing [10, 21, 25], predictions of fault likelihood [6], dynamic code measurement [17], test suite minimization [22, 27], and test case prioritization [4, 23, 26].

Often, code coverage information is collected for a version of a program to aid in some maintenance or testing task performed on that particular version. For example, the execution of test suite  $T$  on version  $P_i$  of program  $P$  generates coverage information that could be used to determine the statement coverage adequacy of  $T$  on  $P_i$ .

In many other cases, however, code coverage information collected on a particular version  $P_i$  of program  $P$  is used to aid in analyses or tasks performed on *subsequent* versions of  $P$ . For example, most regression test selection and test case prioritization techniques (e.g. [4, 10, 21, 23, 25, 26]) use test coverage information from  $P_i$  to help select or prioritize the tests that should be executed on some later version  $P_{i+j}$  of  $P_i$ . Similarly, some techniques for reliability estimation [11] use coverage information from  $P_i$  to assess the risk of executing certain components in  $P_{i+j}$ .

In many such cases, reuse of coverage data is essential. It would make no sense, for example, to run all the tests in  $T$  on  $P_{i+1}$  in order to use that coverage information to select the subset of  $T$  that must be run on  $P_{i+1}$ ! In other cases, it simply is not cost-effective to re-gather coverage information (reapplying expensive profiling techniques and re-executing the program many times) for each successive version of an evolving program. Instead, coverage information is gathered on some version of  $P$ , and re-used — without being recalculated — on several subsequent versions.

Of course, as software evolves, modifications to that software can alter that software's behavior on particular inputs, and code coverage information calculated for a set of inputs on a version  $P_i$  of  $P$  may not accurately reflect the coverage that would be obtained if that set of inputs were applied to subsequent versions. Thus, techniques that rely on previously computed code coverage information may depend for their success on the assumption that coverage information remains sufficiently stable as software evolves.

Despite the importance of code coverage information, and the frequency with which maintenance techniques depend on its reuse, in our search of the research literature

we find little data on the effects of program evolution on coverage information. Rosenblum and Weyuker [20] conjecture that coverage information remains relatively stable across program versions in practice, and this hypothesis is supported in their study of regression test selection predictors. Studies of techniques that rely on the relative stability of coverage information [4, 23] have shown that those techniques can succeed, suggesting indirectly that sufficient stability may exist. Beyond these informal or indirect reports, however, we can find no previous research specifically addressing questions about the impact of software evolution on code coverage information.

We are therefore conducting empirical studies investigating code coverage and its stability in evolving software. This paper reports the results of two such studies: a controlled experiment and a case study. Our results indicate that even small changes during the evolution of a program can have a profound impact on coverage information, and that this impact increases rapidly as the degree of change increases. Furthermore, our results suggest that the impact of evolution on coverage information may be difficult to predict. These findings have consequences for certain techniques that use code coverage information from earlier software versions to perform tasks on subsequent versions.

In the next section of this paper we present our research questions and measures, and discuss our empirical approaches. Sections 3 and 4 present our two studies in turn, describing their design and results. Finally, Section 5 discusses the overall implications of the results of both studies, and discusses future work.

## 2 Empirical Studies

We are interested in the following research questions:

**RQ1:** How does program evolution affect code coverage information?

**RQ2:** What impact can a code modification have on code coverage information?

**RQ3:** Are certain granularities of code coverage information more stable than others?

### 2.1 Measures

We can represent the execution of a test suite<sup>1</sup> on a program by a coverage matrix  $C(v)$ . The number of rows  $c$  in this matrix equals the number of program components for which coverage is being tracked, and the number of columns  $t$  equals the number of tests in the test suite. The

<sup>1</sup>Coverage information can be gathered for any set of inputs to a program and used in analyses. Such sets of inputs might be test suites, but they might also be samples drawn on operational profiles, or any other collections of inputs of interest. For simplicity, in this paper, we use the term “test suite” to describe any such set.

value of a cell in the matrix can be 1 or 0, depending on whether a component was covered by a test or not.

For example, Table 1 depicts two versions of the function `computeTax` ( $v_0$  and  $v_1$ ). The changes between versions are shown in italics. Table 2 shows a test suite developed to achieve statement coverage of version  $v_0$  of the function, listing its coverage of statements in both versions, and Table 3 presents the coverage matrices,  $C(v_0)$  and  $C(v_1)$ , that result from executing that test suite on the two versions.

To quantify differences in coverage information, and to let us measure the effects of evolution on coverage, we selected four metrics. Let  $C(v_i)$  and  $C(v_j)$  be coverage matrices for versions  $i$  and  $j$  of a program, respectively.

- **Matrix density (MD)** measures the distribution of a test suite across a set of components. MD is computed by counting each 1 in  $C(v_i)$ , and then dividing by  $c \times t$  and multiplying by 100. For our sample program MD is 62% for  $C(v_1)$  and 48% for  $C(v_2)$ .
- **Component coverage (CC)** measures the percentage of components executed by a test suite. CC is computed by counting each component that was executed by at least one test, and then dividing by  $c$  and multiplying by 100. For our sample program CC is 100% for  $C(v_0)$  and 86% for  $C(v_1)$ .
- **Change across components (CAC)** measures the percentage of change in component coverage between coverage matrices  $C(v_i)$  and  $C(v_j)$ . CAC is computed by counting the number of components that did not receive identical coverage on both versions (vector comparison), dividing it by  $c$ , and multiplying by 100. For the two versions of our sample program, CAC is 71%.
- **Change across tests (CAT)** measures the percentage of change in test suite execution between coverage matrices  $C(v_i)$  and  $C(v_j)$ . CAT is computed by counting the number of inputs that did not execute the same components on both versions, dividing it by  $t$ , and multiplying by 100. For the two versions of our sample program, CAT is 67%.

These metrics capture different aspects of code coverage information. The first two metrics relate to individual coverage matrices, and quantify the relationship of a test suite to a set of components. The second two metrics relate to changes in coverage between versions, and quantify the extent to which that coverage changes.

### 2.2 Empirical Approaches

To investigate our research questions, we need to measure how coverage information is affected as a program evolves. A program evolves as a result of changes (modifications, enhancements, adaptations). Hence, to study the effects of program evolution on code coverage we require programs with changes.

Statement	Version $v_0$	Statement	Version $v_1$
s1	computeTax (float base, char marital_status) {	s1	computeTax (float base, char marital_status) {
s2	if (base > 5000.0){	s2	if (base > 10000.0){
s3	if (marital_status == 's')	s3	if (marital_status == 'm')
s4	tax = base * 10.0;	s4	tax = base * 10.0;
s5	else	s5	else
s6	tax = base * 15.0;	s6	tax = base * 15.0;
s7	tax = tax + adjustment; }	s7	tax = tax + adjustment; }

**Table 1. Sample program versions  $v_0$  and  $v_1$ .**

Test	Input Values	Statements Covered	
		$v_0$	$v_1$
t1	base = 99999.0, marital_status = 's'	1,2,3,4,7	1,2,3,5,6,7
t2	base = 5001.0, marital_status = 'm'	1,2,3,5,6,7	1,2
t3	base = 4999.0, marital_status = 's'	1,2	1,2

**Table 2. Test suite for sample program.**

	$C(v_0)$			$C(v_1)$		
	t1	t2	t3	t1	t2	t3
s1	1	1	1	s1	1	1
s2	1	1	1	s2	1	1
s3	1	1	0	s3	1	0
s4	1	0	0	s4	0	0
s5	0	1	0	s5	1	0
s6	0	1	0	s6	1	0
s7	1	1	0	s7	1	0

**Table 3. Coverage matrices for  $v_0$  and  $v_1$ .**

One empirical approach is to perform a case study on an existing program that has several versions and existing test suites. The changes to such a program would be “real”. However, under this approach, the circumstances under which evolution and testing occur are not controlled. For example, the changes may have different sizes and implications, and there may be only one test suite per version. Hence, it could be difficult to draw valid inferences about causality or about the ability of results to generalize.

A second empirical approach is to perform a controlled experiment in which changes are simulated. The advantage of such an experiment is that the independent variables of interest (number of changes, location of changes, and test suite) can be manipulated to determine their impact on dependent variables (MD, CC, CAC, CAT). This lets us apply different values to the independent variables in a controlled fashion, so that results are not likely to depend on unknown or uncontrolled factors. The primary weakness of this approach, however, is the threat to external validity posed by the change simulation.

Because each approach has different advantages and disadvantages, we employed both.

### 3 Study 1

We present our controlled experiment first.

#### 3.1 Experiment Instrumentation

**Subject Program and Tests.** As a subject for this experiment, we used a C program developed for the European Space Agency, containing 6218 lines of executable code; we refer to this program as *Space*. *Space* is an interpreter for an array definition language (ADL). *Space* reads a file of ADL statements, and checks the contents of the file for adherence to the ADL grammar and consistency rules. If the file is correct *Space* outputs a list of array elements, positions, and excitations; otherwise it prints error messages.

Initially, we constructed a pool or “universe” of tests for *Space* in two phases. We began with 10,000 randomly generated tests created by Vokolos and Frankl [24]. Then we created new tests until each executable branch (outcome of a decision statement) in the program was exercised by at least 30 tests. This process yielded a pool of 13,585 tests.

To obtain sample test suites, we used the pool of tests, and coverage information about those tests, to generate 1000 test suites that were branch-coverage-adequate for *Space*. (Note that, because *Space* contains some code that is unreachable, including some unreachable “debugging” functions, there were branches in *Space* not executed by any tests). For our experimentation, we randomly selected 50 of these test suites.

**Modeling the Effects of Changes.** To study the impact of changes on *Space*, we instrumented each *if* and *while* statement in the program. The instrumentation consisted of a call to a change-probability function. The inserted function used a probability distribution to determine whether or not an instrumented branch should be changed.<sup>2</sup> A negation of the evaluated condition constituted a change. This procedure allowed us to generate different versions by creating new change probability distributions.

To understand the impact of the amount of change on code coverage information, we investigated five levels of

<sup>2</sup>Note that each instrumented branch was associated with a probability of being changed. This probability was generated randomly before compilation and checked at execution time by the change-probability function.

change: 1%, 2%, 5%, 10% and 15%. For example, at the 1% change level, approximately six of the 513 instrumented branches of `Space` had a non-zero probability of being changed. We generated 30 versions for each change level, each with a randomly defined change probability.

**Types of Code Coverage Information.** One of our research questions concerns differences in types of code coverage information. For this experiment, we selected two types of coverage: *statement coverage*, which tracks the statements executed by each test, and *function coverage*, which tracks the functions executed by each test. In both cases, we track not frequency of execution, but just whether the component was or was not executed. To measure coverage, we used the Aristotle analysis system [9].

**Additional Infrastructure.** Some branch modifications caused `Space` to crash. One option for handling this was to discard tests that caused crashes; however, this could discard valuable coverage information, causing us to underestimate the effects of changes. Moreover, in practice, such tests would not be discarded. A second option was to modify `Space` to capture termination signals and record coverage data before failing. We chose this second option. Similarly, we inserted traps into `Space` to halt execution on modifications that lead to infinite loops. (In practice, such loops would be terminated by human intervention, our traps simulate this.)

### 3.2 Experiment Design

To put `Space` and its modifications through all desired experimental conditions we selected a factorial design, considering all combinations of versions, test suites, and change levels. In more formal terms, our design constitutes a completely randomized factorial design with three treatments: (T1) versions, with 30 levels (excluding the baseline), (T2) test suites, with 50 levels, and (T3) change level, with five levels. Given this design, we generated 7500 observations (excluding the 50 observations for the baseline) at the statement level and 7500 observations at the function level.<sup>3</sup> Each observation contained the four metrics presented in Section 2.1, for each level of the three factors.

### 3.3 Results and Analysis

First, we examine the data graphically. Box plots for each metric at the baseline and the five change levels are presented in Figure 1.<sup>4</sup> The column of graphs on the left

<sup>3</sup>To address our third research question, we joined both data sets adding a new treatment for coverage type (T4 with 2 levels).

<sup>4</sup>Box-plots summarize the distribution of a variable by using three components: (1) a central line to indicate central tendency or location, (2) a box to indicate variability around this central tendency, (3) whiskers around the box to indicate the range of the variable. In our case, we choose the means, standard errors and standard deviations as these three components.

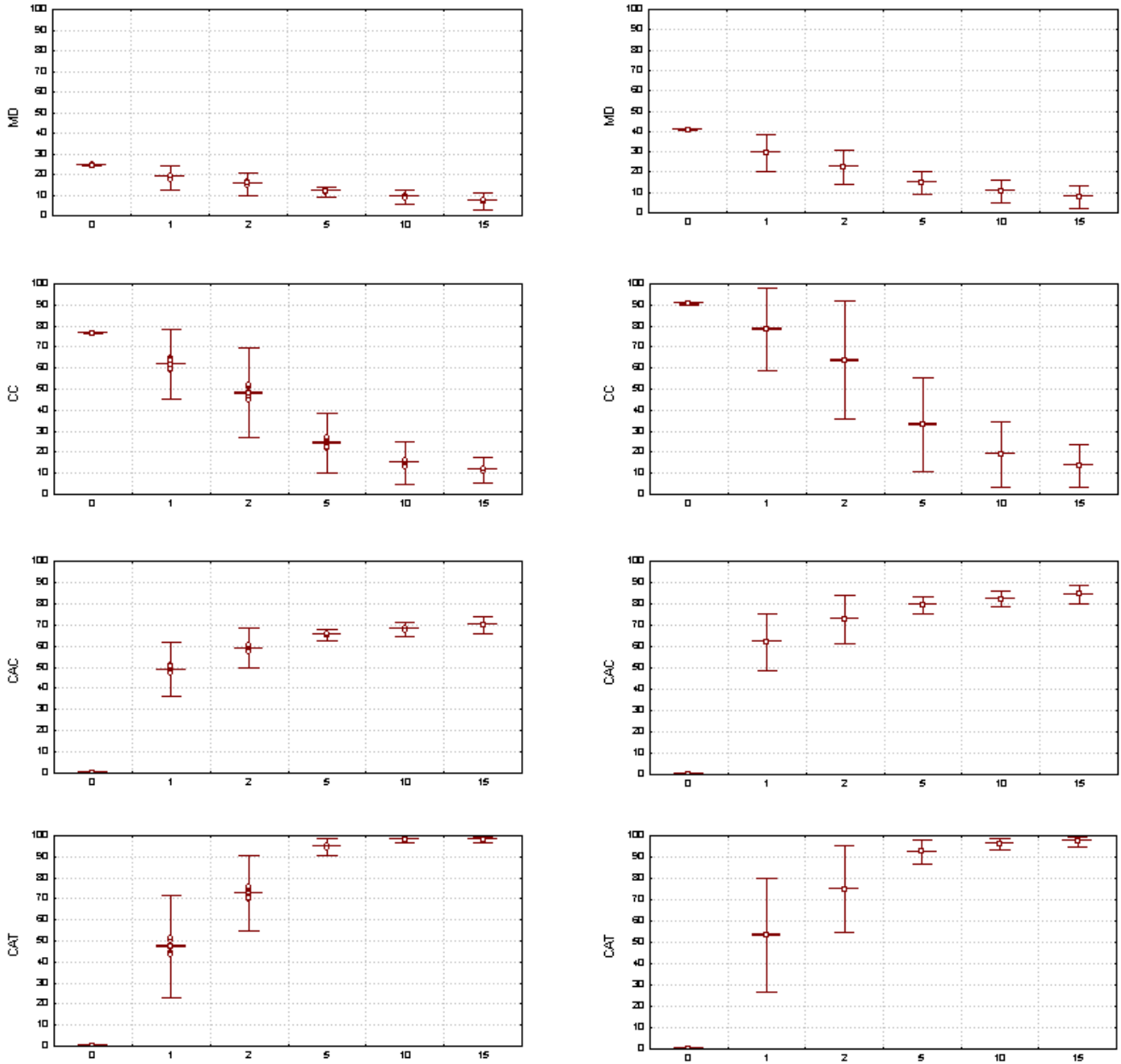
depicts results for statement coverage; the column on the right depicts results for function coverage; the four graphs in each column depict results for each of the four metrics (MD, CC, CAC, CAT), respectively. The x-axis represents the baseline and five change levels. The y-axis represents the values measured for each metric. Each individual box plot (other than the plots for baselines) indicates the distribution of the 1500 observations (50 test suites times 30 versions) for one change level, for a given metric and coverage type. (Plots for baselines, where no change levels are involved, represent 50 observations.)

As the plots indicate, MD (matrix density) decreased as change level increased. Both statement and function coverage presented the same trend. However, MD based on function coverage information exhibited a greater rate of decrease and greater deviation (as indicated by the size of the box plots). CC (component coverage) also decreased as change level increased. As expected, CC at the function level was higher at lower change levels, but at higher change levels it rapidly decreased to values similar to CC at the statement level. (Recall from Section 3.1 that `Space` contains some unreachable code and functions, this explains why CC is not 100% for either type of coverage, even for the baseline version.)

It is interesting to note that the mean CC decreases more than 10% if even 1% of the branches are affected by a change. That decreased percentage rose to 20% when 2% of the branches were affected by a change. In other words, a minor change can have a profound effect on the coverage structure as captured by the CC measure. It is also worth noting that the MD and CC metrics seem to stabilize as change levels increase (but at smaller values). Finally, the expected CC at lower change levels is more difficult to predict than at higher change levels. This is evident in the large deviation present in the box plots at lower change levels, which means that the impact on CC at lower levels is conditioned on other factors (e.g., the type of change that is made). For higher change levels, there is more certainty that CC will decrease significantly, independent of other factors.

The change metrics, CAC (change across components) and CAT (change across tests), display an opposite trend. These metrics reflect the degree of change in coverage information as program change level increases. CAC using function coverage information seems to be impacted the most by changes in the program: a 1% change in the program has an average impact of over 60% on CAC based on function coverage information. The same tendencies can be observed in CAT; however, the CAT values presented more variability than the CAC values at lower levels of change. Also, for change levels above 5%, the CAT average was over 90% for both statement and function coverage.

To corroborate the trends observed in the graphs, we tested for differences across change levels between the



**Figure 1. Box plots showing the distribution of coverage measures observed in Study 1.**

means of all metrics. The appropriate procedure for this type of test involving several means is an analysis of variance (ANOVA) [13]. The ANOVA calculations were performed using STATICA, a commercial package that facilitates this type of statistical analysis.

The null hypothesis states that the means at all change levels are the same for each of the metrics. Since there are four metrics, we performed 4 analyses of variance comparing all the change levels. Table 4 summarizes the results of these analyses, at both the statement and function coverage levels. The summary includes the degrees of freedom of

each effect and the p-value of each metric. For our experiments, a p-value less than 0.05 indicates that the effect is significant (so we should reject the null hypothesis).

We consider five effects: (1) change level, (2) version, (3) test suite, (4) interaction between change level and version, and (5) interaction between change level and test suite. Although we are investigating primarily the impact of changes on coverage, understanding these other effects can help us judge whether other factors affect our results.

Where statement coverage information is concerned (columns three through six in Table 4), the means of all

Effect	Degrees of Freedom	p values for metrics at statement level				p values for metrics at function level			
		MD	CC	CAC	CAT	MD	CC	CAC	CAT
Change Level	4	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Version	29	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Test Suite	49	0.014	1.000	1.000	0.405	0.000	1.000	0.998	0.746
Change Level * Version	116	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
Change Level * Test Suite	245	0.000	0.962	1.000	0.000	0.000	0.945	0.984	0.001
Error	7100								
Total	7544								

**Table 4. Anova analysis of the four metrics at the statement and function levels.**

Change Level	statement level								function level							
	MD		CC		CAC		CAT		MD		CC		CAC		CAT	
	Mean	Grp	Mean	Grp	Mean	Grp	Mean	Grp	Mean	Grp	Mean	Grp	Mean	Grp	Mean	Grp
0	24.51	A	76.56	A	0.00	A	0.00	A	40.79	A	90.37	A	0.00	A	0.00	A
1	18.41	B	61.61	B	48.97	B	47.24	B	29.33	B	78.48	B	62.19	B	53.29	B
2	15.15	C	48.04	C	58.81	C	72.67	C	22.34	C	63.64	C	72.68	C	74.92	C
5	11.62	D	24.39	D	65.33	D	94.49	D	14.76	D	33.19	D	79.25	D	92.48	D
10	9.15	E	14.85	E	67.81	E	97.79	F	10.75	E	19.09	E	82.26	E	96.19	E
15	7.05	F	11.41	F	69.86	F	97.99	F	7.94	F	13.83	F	84.48	F	97.15	F

**Table 5. Bonferroni analysis of the four metrics at the statement level and function levels.**

metrics were affected by change level, version, and the combination of both. This is intuitively plausible: the amount and location of the change affects all the coverage metrics. The interaction between change level and versions is significant, indicating that further per-contrast analysis is necessary to understand the interaction. The test suite has a significant effect only on MD, which is reasonable given that all the test suites were branch coverage adequate. A similar scenario occurs for the interaction between test suite and change level, although their combination also affects CAT. The sensitivity to changes of MD and CAT may have been the cause for obtaining significance on all effects.

Where function coverage is concerned (rightmost four columns), the results are similar to those for statement coverage. The same effects are significant across all metrics: change level, version, and the interaction of both.

For all metrics, the null hypothesis was rejected for the change level effect (not all means for all change levels are the same). To determine which change levels actually differed, we performed a Bonferroni analysis [13] (see Table 5). For each metric, the table presents the mean, and a grouping letter to represent the Bonferroni results (change levels with the same letter are not significantly different).

Although the averages for all metrics seem to grow closer as change level increases, we found that only in one case (comparing CAT at the 10% and 15% levels for statement coverage) are the metrics not significantly different.

Finally, we performed an analysis to compare the metrics means between types of coverage information (see Table 6).

The ANOVA indicated that, when compared at each change level, statement and function coverage information generated significantly different metrics. As expected, most of the comparisons indicated that function coverage information tends to have higher values across all change levels for MD and CC. This is intuitively plausible when we consider that it is common for a change that alters statement coverage not to modify the functional control flow.

The interpretation of the change metrics was not as intuitive. For CAC, the analysis indicated that function coverage information was more susceptible to changes in the program than was statement level information. That means that although CC may have remained similar between versions, the functions that were covered were different. CAT on the other hand shows different results depending on the change level. At the 1% and 2% levels, function coverage information follows the same trend as CAC. This trend, however, is reversed at higher change levels, possibly reflecting different thresholds on maximum change.

### 3.4 Threats to Validity

**Internal.** The infrastructure required to execute this study involves many tools. Although many of the tools were used and tested in previous experiments, some new tools were written to gather, filter and process coverage information. We manually verified results and ran the study repeatedly to obtain confidence in the new tools' correctness.

The instrumentation of the subjects constitutes a threat in itself. We performed several checks on the code to en-

Metrics	baseline		1%		2%		5%		10%		15%	
	st	f	st	f	st	f	st	f	st	f	st	f
MD	24.514	40.794	18.413	29.332	15.151	22.336	11.619	14.760	9.155	10.754	7.047	7.940
CC	76.560	90.367	61.608	78.477	48.038	63.636	24.393	33.186	14.849	19.099	11.413	13.828
CAC	-	-	48.975	62.198	58.810	72.682	65.327	79.253	67.807	82.258	69.862	84.476
CAT	-	-	47.238	53.291	72.670	74.921	94.495	92.483	97.792	96.192	97.998	97.149

**Table 6. Metrics means for different types of coverage information across different change levels.**

sure that we had introduced only the code needed to capture the necessary data. This process became more complicated than expected when the program began behaving unreliably (crashing or entering infinite loops). Additional control instrumentation was inserted to manage this problem.

**Construct.** We controlled the independent variables by using different versions, test suites and change levels. However, we found that there was significant interaction between test suite and change level for some metrics that could be attributed only to the sensitivity to change of those metrics, which indicates that they may not be very robust.

Our dependent variables capture most of the information in any coverage matrix; however, there are types of coverage matrices we did not consider (e.g., for class coverage). Nevertheless, we believe that the coverage information and metrics used appropriately reflect the notion of coverage.

**External.** In this controlled study, the definition of a change is the negation of a branch and the location of the change (and the probability of changing) is determined by a probability function. Both of these items — the definition and the location/probability of a change — may not represent the effects of the universe of changes possible during program evolution. On the other hand, many other types of changes (e.g., changes to variable assignments, additional control flow structures, etc.) result in effects on branch conditions, and these effects may be modeled by changes to those conditions. Thus, our changes constitute a simplification of the results of many other type of changes.

Also, modifications might not just alter control flow, they might add new elements to it. In this study, we omitted that type of modification so as to preserve the ability to compare coverage information across versions. However, additions to the control flow structure would be expected to have an even greater impact on the decay of coverage information, which further supports our conclusions.

The study was conducted on one program (*Space*), and factors specific to that program such as its structure, architecture, and size may have had an impact on the results. Although the study of additional subjects would be useful, we believe that we limited the impact of this threat by using 30 versions per level. Since the version generation involved a random based change probability distribution, the program structure was randomly affected, which reduces the threat.

## 4 Study 2

In our first study, the major threats to validity were external. Specifically, the conclusions that could be drawn from the results may have been limited by the representativeness of the subject and the manner in which changes were introduced to generate different versions.

Our second study addresses some of these validity questions by examining the effects of evolution on a large application, across several actual versions, using its own set of test suites, and including a wider range of types of changes (e.g., addition and deletion of code). The study utilizes the same set of metrics as the previous study, and attempts to answer the first two research questions.

### 4.1 Experiment Instrumentation

**Subject Program and Tests.** As a subject program, we used *Bash*, a C program developed as part of the GNU Project [7, 18]. *Bash*, which stands for “Bourne again Shell”, is GNU’s shell, which adds functionality to the Bourne shell. Functionality from the Korn and C shells, as well as new functionality, have been added to each new version of *Bash*. For our study, we used the seven most recent public release versions of *Bash* (versions 2.0 through 2.04). Table 7 describes these versions. For each version, column 2 lists the number of lines of code in that version, column 3 lists the number of functions in that version, and column 4 lists the total number of functions in common between that version and the succeeding versions. It is evident from this table that the lines of code (LOC) and number of functions increased as *Bash* evolved.

Table 8 quantifies the degree of change between versions, listing the number of functions changed from each version listed in the first column to each later version listed along the top.

Each of the *Bash* versions has an associated test suite. Each test suite is composed of tests that focus on different program functionalities. Table 9 presents a short description of the test suites, including the number of tests and statistics summarizing test size in terms of the number of lines (note that each line is made of one or more Unix commands).

**Types of Code Coverage Information.** Due to the size of *Bash*, in this study we measured only function coverage.

Version	LOC	# of functions	# of common functions
2.0	70,622	1503	1352
2.01	72,462	1549	1403
2.01.1	72,605	1550	1404
2.02	82,331	1693	1617
2.02.1	82,358	1693	1617
2.03	83,855	1735	1656
2.04	90,279	1927	-

**Table 7. Bash versions.**

Version	# of functions changed for					
	2.01	2.01.1	2.02	2.02.1	2.03	2.04
2.0	219	236	336	338	389	465
2.01		37	202	205	281	407
2.01.1			183	186	266	400
2.02				12	141	344
2.02.1					135	340
2.03						261

**Table 8. Bash evolution.**

We used the CLIC tool [5] to obtain this coverage information. Because comparisons of data involving differing numbers of functions could obscure the analysis of differences in coverage (e.g., conflating differences due to numbers of functions with differences due to changes in coverage) we considered only functions present in all versions.

## 4.2 Experiment Design

We ran each test suite on its corresponding version and all posterior versions. For example, we ran the test suite corresponding to version 2.0 on all versions, and ran the test suite for version 2.03 on versions 2.03 and 2.04. Using this approach, we were able to generate coverage information for several test suites on multiple versions, making our results less dependent on a particular test suite’s qualities.

## 4.3 Results and Analysis

The results of the case study are synthesized in Figure 2, which depicts four graphs, one for each metric. The x axes represent the versions, in succession from left to right. The y axes represent the values measured for each metric. Within each graph, each test suite is represented by a line that joins the metric values measured for that test suite across the various Bash versions.

As the graphs show, MD tended to decrease as a test suite was applied to posterior versions. That tendency was evident over all test suites. Note that between relatively similar versions, the rate of change was almost null (e.g., between version 2.01 and 2.01.1).

CC exhibited greater differences among test suites. The test suite associated with version 2.0 seemed to be the weak-

Test Suite	tests	Number of lines in each test			
		Mean	Std. Dev.	Max.	Min
2.0	23	90.783	118.123	372	1
2.01	34	126.265	123.210	454	1
2.01.1	34	126.265	123.210	454	1
2.02	38	148.105	126.406	460	5
2.02.1	38	148.105	126.406	460	5
2.03	39	148.538	129.441	462	5

**Table 9. Bash test suites.**

est, providing a maximum of 23.8% CC, while the test suite corresponding to version 2.01 had a CC value of 32.3%. In spite of these differences among test suites, the tendency was the same: the further a test suite was applied, the smaller CC became.

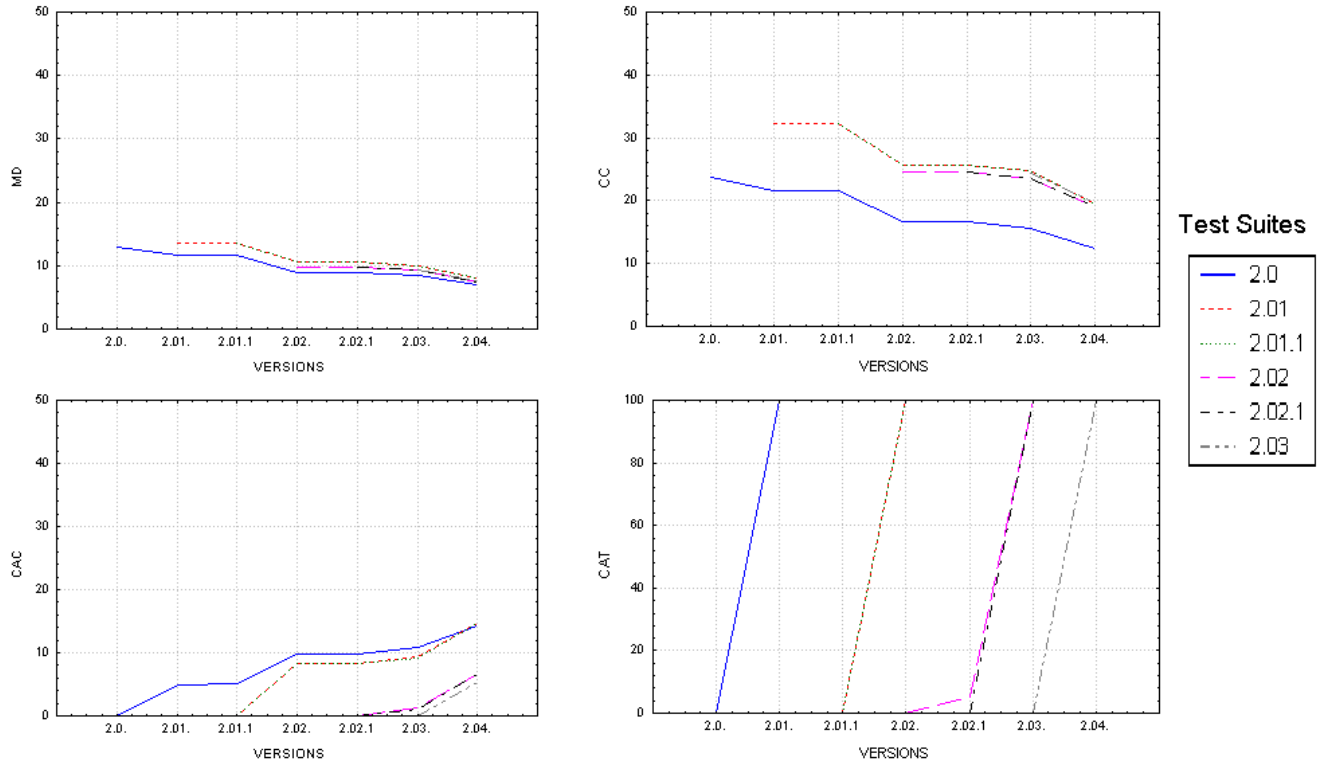
CAC showed the expected increasing tendency, with results independent of the test suite we employed. It is interesting to observe that among the most similar versions, the metric showed limited change. For example, when the test suite for version 2.01 was executed on versions 2.01 and 2.01.1, CAC between those coverage matrices was 0.06%.

CAT presented a more extreme picture. Its values were always either close to 0% or exactly 100%. The values were zero only when the target versions were very similar. When the comparison was performed between other versions, all the tests presented a different execution distribution so CAT was 100%. We found this result a bit disturbing until we realized that the tests for Bash are quite comprehensive (they each cover a large number of the common functions) and unless a change is localized, it will impact the majority of the tests. We observe that this was not the case for the test suite associated with version 2.02 when it is executed between versions 2.02 and 2.02.1, where the change was minimum and CAT was 5.2%.

## 5 Summary and Conclusions

We have presented the results of two studies examining the effects of evolution on code coverage information, and measuring those effects along four dimensions. We used two approaches in obtaining these results. Our first study, a controlled experiment, provided a flexible platform with which to vary independent variables of interest in a controlled manner and observe the effects of that variance. Our second study, a case study using “real” modifications, did not offer flexibility or control; however, because the subject studied may be representative of typical evolving systems, the second study helped address threats to external validity. Together, the relatively consistent results observed in the two studies increase the confidence we can have in our assessments.

As we have discussed, these studies, like any other, have certain limits to their validity (e.g., the representativeness of the selected programs). Keeping these in mind, in this



**Figure 2. Graphs showing Bash coverage information observed in Study 2.**

section we now compare the results of these studies further, and discuss some of their implications.

First, our overall results were consistent across both studies: even small changes during the evolution of a program can have a profound impact on coverage information, and this impact increases rapidly as the degree of change increases. For example, in our first study we discovered that even when only 1% of the branches in a program were affected by changes, mean coverage of program statements was reduced by 16%, and mean coverage of functions was reduced by 10%. A similar reduction in function coverage was also observed for all applicable Bash test suites on four of the six pairs of successive Bash versions examined. This outcome stands in contrast to suggestions in the literature [20] that coverage information remains relatively stable across program versions in practice.

Not surprisingly, both studies also illustrate that the greater the change, the greater the impact on the quality of the coverage information, under all metrics. Of more surprise, however, is the amount of deterioration observed for only relatively small increases in change. Furthermore, our first study suggests that the impact of changes on coverage information can be difficult to predict; this is indicated by the wide range in the values of the various metrics fol-

lowing modifications. These results occurred for coverage measured at the statement and function levels.

Comparisons of the results of the two studies in light of the types of test suites used in each one, however, also suggest that the amount of deterioration in the coverage behavior of individual tests is in part a function of test design. Larger, coarser granularity tests (tests that execute a large proportion of a system’s components) as the ones used in the second study, are likely to exhibit much more extreme degradation in coverage information than smaller granularity tests, at a given level of change.

These results suggest, at a minimum, that assumptions about the stability of coverage information across program versions should be questioned, and that analysis, testing and maintenance techniques that rely on such stability may need to be re-evaluated or reconsidered. This does not imply that all techniques that rely on coverage information are suspect; rather, it implies that the manner in which coverage information is relied on may be important.

To illustrate, consider regression test selection techniques. The technique of Rothermel and Harrold [21] avoids the difficulties caused by decay of coverage information due to its specific algorithm. Informally, this technique relies on coverage information only up to the initial point at

which the execution of a test reaches modified statements, selecting all test cases that reach that point (a formal proof is provided in [21].) On the other hand, techniques such as that presented by Hartmann and Robson [10], which rely on a coverage matrix such as that depicted in Table 3 and a change matrix listing which program components are modified to select *one* test case through each modified program statement, may fail to ensure selection of a test through each component in the modified program.

Clearly, further study of the effects of evolution on coverage information is needed, including studies of the sort we have reported here. Future studies could also compare different types of coverage granularity information, and most important, the performance of the techniques that use coverage information themselves, applied across a variety of evolving programs. The results of this initial research provide a beginning and motivation for such studies.

## Acknowledgements

This work was supported in part by the NSF Information Technology Research program under Awards CCR-0080898 and CCR-0080900 to University of Nebraska, Lincoln and Oregon State University, The work was also supported in part by a NASA-Epscor Space Grant Award to the University of Nebraska, Lincoln, and by NSF Faculty Early Career Development Award CCR-9703108 to Oregon State University. Alberto Pasquini, Phyllis Frankl, and Filip Vokolos provided the *Space* program and many of its tests. Praveen Kallakuri helped with the preparation of *Bash*.

## References

- [1] H. Agrawal and J. Horgan. Dynamic program slicing. In *Conf. Prog. Lang. Des. and Impl.*, pages 246–256, June 1990.
- [2] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Trans. Prog. Lang. Sys.*, 16(4):1319–1360, July 1994.
- [3] S. Bohner and R. Arnold. *Software Change Impact Analysis*. IEEE Computer Society Press, Los Alamitos, CA, 1996.
- [4] S. Elbaum, A. Malishevsky, and G. Rothermel. Prioritizing test cases for regression testing. In *Int'l. Symp. Softw. Test. and Anal.*, pages 102–112, Aug. 2000.
- [5] S. Elbaum, J. Munson, and M. Harrison. CLIC: A tool for the measurement of software system dynamics. In *SETL Technical Report - TR-98-04.*, 04 1998.
- [6] S. G. Elbaum and J. C. Munson. Evaluating regression test suites based on their fault exposure capability. *J. Softw. Maint.*, 12(3):171–305, 2000.
- [7] B. Fox and C. Ramey. *bash - Shell of the GNU operating system*. <http://www.gnu.org/gnulist/production/bash.html>.
- [8] T. Goradia. Dynamic impact analysis: A cost-effective technique to enforce error-propagation. In *ACM Int'l. Symp. on Softw. Test. and Anal.*, pages 171–181, June 1993.
- [9] M. Harrold and G. Rothermel. Aristotle: A system for research on and development of program analysis based tools. Technical Report OSU-CISRC- 3/97-TR17, Ohio State University, Mar 1997.
- [10] J. Hartmann and D. Robson. Revalidation during the software maintenance phase. In *Conf. on Softw. Maint.*, pages 70–79, Oct. 1989.
- [11] J. Horgan and A. Mathur. Software Testing and Reliability. In M. Lyu, editor, *Handbook of Software Reliability Engineering*, chapter 13. McGraw-Hill, New York, 1995.
- [12] M. Kamkar and P. Krajina. Dynamic slicing of distributed programs. In *Int'l. Conf. Softw. Maint.*, pages 222–229, Oct. 1995.
- [13] R. E. Kirk. *Experimental Design: Procedures for the Behavioral Sciences*. Brooks/Cole, Pacific Grove, CA, 3rd edition, 1995.
- [14] B. Korel and J. Rilling. Dynamic program slicing methods. *Info. and Softw. Tech.*, 40(11–12):647–659, Dec. 1998.
- [15] J. R. Larus. Efficient program tracing. *Softw. Pract. Exper.*, 20(12):1241–1258, Dec. 1993.
- [16] M. Marre and A. Bertolino. Reducing and estimating the cost of test coverage criteria. In *Int'l. Conf. on Softw. Eng.*, pages 486–494, Mar. 1996.
- [17] J. C. Munson and G. A. Hall. Estimating test effectiveness with dynamic complexity measurement. *Emp. Softw. Eng. J.*, 1(3):279–305, 1996.
- [18] C. Newham and B. Rosenblatt. *Learning the Bash Shell*. O'Reilly Associates, New York, NY, 1998.
- [19] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Trans. on Softw. Eng.*, 11(4):367–375, Apr. 1985.
- [20] D. Rosenblum and E. J. Weyuker. Using coverage information to predict the cost-effectiveness of regression testing strategies. *IEEE Trans. on Softw. Eng.*, 23(3):146–156, Mar. 1997.
- [21] G. Rothermel and M. Harrold. A safe, efficient regression test selection technique. *ACM Trans. on Softw. Eng. and Methodology*, 6(2):173–210, Apr. 1997.
- [22] G. Rothermel, M. J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Int'l. Conf. Softw. Maint.*, Nov. 1998.
- [23] G. Rothermel, R. Untch, C. Chu, and M. J. Harrold. Test case prioritization: an empirical study. In *Proc. Int'l. Conf. Softw. Maint.*, pages 179–188, Aug. 1999.
- [24] F. I. Vokolos and P. G. Frankl. Empirical evaluation of the textual differencing regression testing technique. In *Int'l. Conf. Softw. Maint.*, pages 44–53, Nov. 1998.
- [25] L. White and H. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Conf. on Softw. Maint.*, pages 262–270, Nov. 1992.
- [26] W. Wong, J. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Intl. Symp. on Softw. Rel. Engr.*, pages 230–238, Nov. 1997.
- [27] W. E. Wong, J. R. Horgan, S. London, and A. P. Mathur. Effect of test set minimization on fault detection effectiveness. In *Int'l. Conf. Softw. Eng.*, pages 41–50, Apr. 1995.